

CIRCUIT CELLAR[®] ONLINE

An S-7800A/ PIC16F877 Journey

FEATURE ARTICLE

Fred Eady

Part 2: Reviving It Up

Continuing on his journey, Fred forges a path all the way through to using the S-7800A/PIC16F877 Internet Engine as a web server. With an everyday PIC, a C compiler, a tiny firmware protocol stack, and some common components, he shows us how to put them on the Internet. As Fred says, "(There's) light at the end of the Internet tunnel!"



or the last couple of weeks, I've been chasing bits all over the Florida room. The S-7600A/PIC16F877 Internet Engine is purring, and in this installment, I'll take you all the way through using it as a web server. However, before I begin the technical discussion, there are some things I need to fix from last month.

The boot loader has been revised to fix some bugs and make it easier to use. The PCW builds an Intel hex file and appends the whole thing with a semicolon followed by the PIC type the file is intended to load into. That semicolon and the text that followed it drove my boot loader menu crazy. The problem was that the boot loader code didn't know that the characters following the last Intel hex line were bogus. It tried to load and analyze them. So, I simply turned off the PIC's serial receiver in the UART and delayed long enough to allow Tera Term Pro to spout all of the leftover junk characters beginning with the semicolon.

A second potential problem involved the boot code vector area. If for any reason the Tera Term Pro download failed, the boot code vector for the failed program would be retained in the NOPs area. If the S-7600A/PIC16F877 Internet Engine was allowed to restart, it would vector to nonexistent or corrupted code. To avoid this, I rewrite the boot code vector area with NOPs if the download fails. This allows the boot loader to loop and wait until valid code and a valid boot code vector are loaded. I also cleaned up the boot loader listing to make it easier to follow. The results can be seen in Listing 1.

TERA TERM PRO

This terminal emulator adds class and a bit of automation to the boot loader PIC code. I assembled a simple macro program to run under Tera Term Pro (see Listing 2), which guides you through the S-7600A/PIC16F877 Internet Engine file upload process. Basically, the Tera Term Pro macro (*upload.ttl*) connects to the S-7600A/PIC16F877 Internet Engine via a personal computer's COM2 serial port and uploads code to the PIC16F877 by way of the boot loader code.

Let's analyze the macro beginning with the definitions area. Tera Term Pro uses message boxes that look like their standard Windows counterparts to communicate with you. The *boxtitle* defines the text that resides in the title area of the message box. The

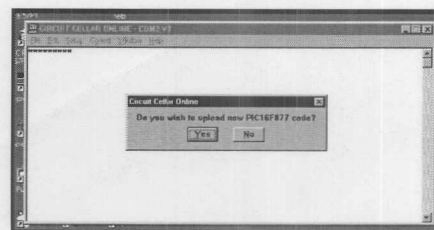


Photo 1—Note the box title. It was all done without having to write a single byte of Windows code.

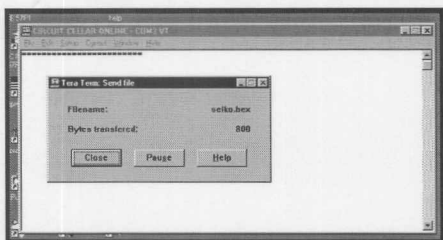


Photo 2—If you're wondering what the asterisks are for, they're just there to let me know the code is running.

rest of the definitions in this area are the wording that will appear in message boxes called from various parts of the macro. The only exception is *uploadfile*, which is actually the Intel hex file that is loaded and executed. In my macro, this file is called *seiko.hex* and is actually the web server code.

Moving on to the main macro code area, you find the label *:begin* followed by a connect command. The */C=2* tag says connect to the personal computer's COM2 serial port. The connect command returns a result code. If the result code is 0, a link to Tera Term Pro has not been made. A result code of 1 means the link to Tera Term Pro is good but there is no connection to the host. You want a result code of 2, which tells you that the link to Tera Term Pro and the connection to the host are up.

Result codes of 0 and 1 trigger a *messagebox* event that informs you of the error and disconnects Tera Term Pro. A result code of 2 causes the macro code to branch to the *:goodlink* label. A *yesnobox* is triggered asking if you would like to upload code to the S-7600A/PIC16F877 Internet Engine.

If you glance at the boot loader listing, you'll see that an ASCII "u" activates the upload portion of the code. A *yes* tells Tera Term Pro to send the ASCII "u" to the boot loader code

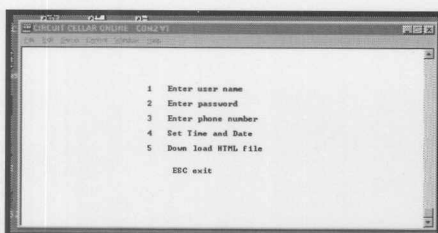


Photo 4—At this point, Tera Term Pro and the S-7600A/PIC16F877 Internet Engine boot loader have completed their tasks and turned over control to the uploaded code.

and wait for the outcome of the upload process. The S-7600A/PIC16F877 Internet Engine boot loader code will send an ASCII "g" if the upload is good and an ASCII "E" if any of the checksum calculations don't match the original line-by-line checksums.

As you can see in the macro listing, a "g" is result code 1 and the "E" is result code 2. This is determined by the order of the ASCII characters behind the *wait* command. A *no* answer to the upload question simply keeps Tera Term Pro connected and nothing else. The boot loader will time out and attempt to run the boot vector code.

ROAD TEST

Now that you know what the Tera Term Pro macro is supposed to do, let's put the code to the test.

Photo 1 is the result of a good connection between Tera Term Pro and the S-7600A/PIC16F877 Internet Engine. The *yesnobox* command was executed and the *uploadprompt* message is displayed. Let's select Yes.

The macro branches to *:loadcode* and immediately executes a *send* command that transmits an ASCII "u." At this point, the boot loader code listens for characters coming into the PIC's serial port. Tera Term Pro issues a *sendfile* command that transfers the *seiko.hex* file to the S-7600A/PIC16F877 Internet Engine. The "0" tag on the *sendfile* command tells Tera Term Pro to send the file as is without modification of the carriage return/line feed sequences. Photo 2 is what you'll see. Serialtest Async gives you a view of what the S-7600A/PIC16F877 Internet Engine and the Tera Term Pro components are doing in Photo 3.

When all of the data in the Intel hex file *seiko.hex* is transferred to the S-7600A/PIC16F877 Internet Engine, an upload complete message box is generated by Tera Term Pro. Click "OK" and Photo 4 is the result. The uploaded code is being executed and is ready to communicate with other components on the S-7600A/PIC16F877 Internet Engine. Before I move on to this phase, I'll impart some knowledge of the hardware to you.

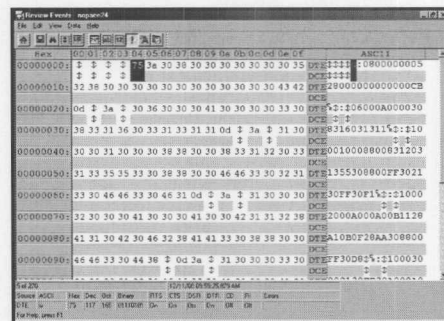


Photo 3—Tera Term Pro is the DTE device and the S-7600A/PIC16F877 Internet Engine is configured as a DCE device. Note the status of the modem control signals at this point.

THE HARDWARE

The heart of the S-7600A/PIC16F877 Internet Engine is the S-7600A. The S-7600A contains all of the necessary hardware and firmware to implement Internet protocols such as TCP/IP, UDP, and PPP. In addition to 10 KB of on-chip RAM to support the protocol stack, the S-7600A has its own UART. Support circuitry for the S-7600A consists of a 74HC4040 that divides the incoming 7.3728-MHz processor clock by 32 and a Sipex SP3243E RS-232 converter.

Data and command information is supplied to the S-7600A by a Microchip PIC16F877 microcontroller. The S-7600A can take information from the PIC serially or with an 8-bit parallel configuration. Because the PIC16F877 has a wealth of I/O, I chose the parallel attachment method. The PIC is clocked at 7.3728 MHz and provides clocking for the S-7600A through the 74HC4040, which presents 230 KHz to the S-7600A's clock input. A Sipex SP3243E RS-232 converter IC that allows the PIC's UART to interface with Tera Term Pro run-

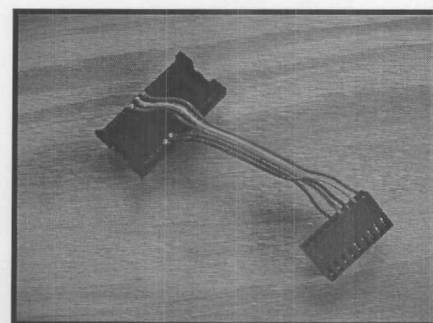


Photo 5—These two signal lines, power and programming voltage lines, are standard fare for programming PICs.

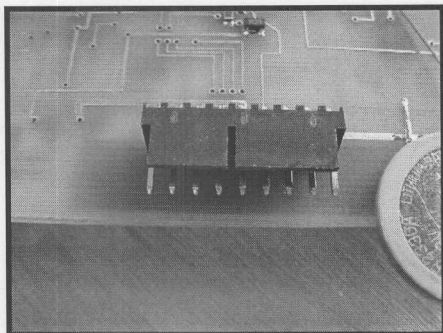


Photo 6—The extra pins are connected to uncommitted I/O lines on the PIC16F877.

ning on a PC also supports the PIC16F877.

Boot loader code is initially loaded onto the PIC16F877 using a standard PIC programmer and a header/connector ribbon cable assembly like the one shown in Photo 5. My PIC programmer uses wide Aries ZIF sockets so I can make my programming jig using a standard 0.600 header. The connector you end up with depends on your PIC programmer setup.

The important thing is to have the necessary signals and voltages between the header and S-7600A/PIC16F877 Internet Engine for programming the PIC16F877. The business end of the in-circuit programming connector on the S-7600A/PIC16F877 Internet Engine is shown in Photo 6.

If I used the space wisely, the PIC16F877 has enough on-chip data EEPROM to accommodate the parameters needed to make a connection to the Internet. However, there wouldn't be enough space left to put a decent HTML image in this memory area. So, a Microchip 24LC256 EEPROM is employed to hold all of the text necessary to connect to the Internet and serve an HTML-based page.

Just to make things interesting, I decided to throw in a Dallas DS1629 time and temperature IC. All the DS1629 needs to operate is a standard 32.768-KHz crystal and some room on the I²C bus. The DS1629 comes hard addressed and uses the extra pins that would normally be address lines for crystal and alarm connections. The 24LC256 is addressed as 0x00 and the DS1629 is addressed at 0x07. The DS1629 clock is volatile and must be backed up with a 3-V lithium cell if you expect to lose power to your S-

7600A/PIC16F877 Internet Engine after it is deployed. And, speaking of power, a National LM2937 3.3-V regulator and a standard 9-VDC power brick powers the whole thing. You can see how it all fits together by perusing the schematic shown in Figure 1. The real McCoy is shown in Photo 7.

AS A WEB SERVER

The S-7600A's protocol stack allows the S-7600A/PIC16F877 Internet Engine to play many roles. In addition to a web server, the S-7600A/PIC16F877 Internet Engine can be configured as a TCP/IP client or an e-mail generator. In this section, I'm going to show you how the S-7600A/PIC16F877 Internet Engine can be programmed to serve a simple web page.

The process begins with the menu you see in Photo 4. After the username, password, and ISP phone number are entered, the date and time can be set and an HTML page can be loaded into the serial EEPROM. I've supplied a sample HTML page in Listing 3. The time and date are taken from the DS1629 and inserted into the page just before it is served.

Listing 4 is the code necessary to implement the web server application on the S-7600A/PIC16F877 Internet Engine. Let's start at the top and work our way down. As an experienced PIC programmer, I found the PCW C package refreshing. The engineers at Microchip were good enough to provide most of the supporting *include* code that does the housework. All of the EEPROM, S-7600A, and PIC16F877 *include* files were provided by Microchip. So, building up the S-7600A/PIC16F877 Internet Engine and the accompanying software was like working with Lego blocks. The remainder of *include* files are the standard C includes with PIC accents that come with the PCW compiler.

I know how to write all of those I²C and serial drivers, but it's great to have them already written for you. As you can see in Listing 4, the serial and I²C ports are defined with a simple line of code for each. The *standard_io* declarations help avoid a common

problem with PIC I/O. PIC I/O is so fast that reads and writes to the same pin can sometimes overlap. The use of *standard_io* makes sure this won't happen by setting the I/O pin for either input or output before the operation is performed. This operation gives the I/O pin time to settle and assures a good read or write to that pin. The rest of the pre-main area is standard stuff. The normal PIC pin definitions are followed by the variable defines.

BRANCHING OUT

If you branch to the Menu function, you'll find that this function is a big switch statement that vectors the program execution to various other functions defined in the pre-main area. For instance, the first selection in Photo 4 calls *Get_username()* and so forth. There are also some routines that directly affect the S-7600A. If you take a look at the S-7600A pins C86 and PSX on the schematic, you'll see they are tied high. This puts the S-7600A in 68K MPU mode. If you compare the S-7600A datasheet timing diagram with the code in the *W_Putc* and *S_Putc* routines, you'll see that the code implements the time transitions found in the datasheet.

For this application, Tera Term Pro is set up to emulate a standard VT-100 terminal. For those of you familiar with DEC equipment, the VT-100 escape sequences in the pre-main functions will be familiar.

The main program begins execution with the initialization of the PIC I/O ports. Because I'm using PORTA as a digital I/O port, all of the analog capability of PORTA is disabled. Timer1 is

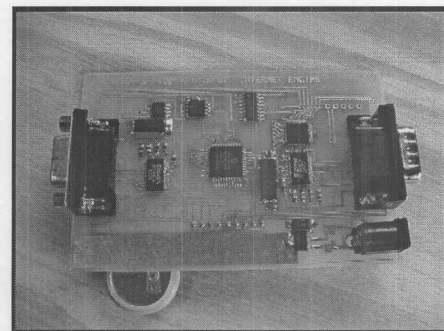


Photo 7—This is the first pass of the circuit board. I'll officially make room for the lithium cell on pass two.

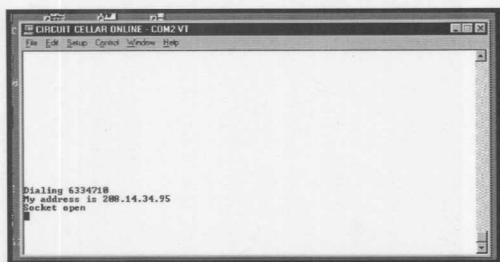


Photo 8—The IP address is the address given to the S-7600A by my ISP during PPP negotiation. Socket open indicates that the S-7600A has passively opened a socket for listening on Port 80.

enabled for 8-μs cycles and used as a general-purpose connection timer. The clock and data lines for the EEPROM and DS1629 are floated, and the DS1629 is instructed to keep time and wait for requests for temperature from the PIC16F877 before beginning a temperature conversion.

If the username, password, phone number, and HTML have been entered previously, a push of the Esc key performs a hard reset on the S-7600A and a DTR reset on the external US Robotics 28.8 modem. The DTR reset is performed by writing 0x06 to the S-7600A's serial port configuration register.

The next task is to program the S-7600A internal clock divider to obtain a 1-KHz internal clock that is used for S-7600A internal affairs. The clock divider register contents are obtained by the following formula:

$$\frac{\text{clock frequency}}{1 \text{ KHz}} - 1 = \text{divide count}$$

The clock frequency is the actual frequency supplied to the S-7600A. In this case, that is 7.3728 MHz through a 74HC4040 that divides by 32, which equates to approximately 230 KHz. Doing the math, that gives us a divide count of 0x00E5.

MAKING CHANGES

In Part 1, I mentioned that I would like to eventually place a Cermetek modem module at the S-7600A UART interface. The Cermetek modem of choice is the CH1786ET, which runs at 2400 bps maximum. The reasons for choosing 2400 bps are:

- The S-7600A/PIC16F877 Internet

Engine won't be sending large chunks of data.

- 2400 bps takes little time to establish ISP connectivity compared to 56-kbps modems.
- The Cermetek modem runs at this speed.
- You can use almost any external modem for initial testing.

So, the S-7600A baud rate divisor can be calculated like this:

$$\frac{\text{clock frequency}}{\text{data transfer rate}} - 1 = \text{data transfer rate divisor}$$

A simple substitution of known values gives you a data transfer rate divisor value of 0x005F.

Most of the time, the modem signals are rigged to simulate desired signal levels by crossing active pins with pins expecting activity on the DB shell connector. I chose to implement the S-7600A's modem control signals correctly because I may want to control data flow in the standard manner. The S-7600A serial port configuration register allows me to directly control DTR and RTS. There's also a bit in the serial port config register that determines who has control of the S-7600A UART and serial port.

At this point, you aren't ready to give control over to the S-7600A. The first order of business is to dial the ISP and do the PPP thing. So, 0x00 is sent to the S-7600A serial port configuration register and DTR and RTS are activated. The US Robotics modem has status LEDs and when the *WriteSeiko(Serial_Port_Config,0x00)* line is executed, LEDs on the modem for DTR and RTS illuminate. Nothing fancy is needed as far as modem configuration is concerned, so an *ATeF* is sufficient to set the modem straight.

READY...SET?

The modem is ready, but before you fire off any phone numbers, the S-7600A must be told that PPP mode will be used and the authentication method will be PAP. Then all of the data entered into the EEPROM via the Menu function in Photo 4 (with the exception of the phone number) is retrieved and placed into the appropriate S-7600A registers. The username

and password are written into the S-7600A PAP string register. The length of the field precedes the actual data. For instance, if the username is Fred, a 0x04 is written to the PAP register first followed by Fred in consecutive ASCII characters. When all of the user data is entered, a final NULL character (0x00) is written to the PAP string register.

The next step is to issue the ATDT modem command followed by the ISP phone number entered earlier. The PIC16F877 has control of the S-7600A UART and is responsible for issuing the dial command. With some extra command work up front, you can configure the US Robotics modem to give you result codes during the dial process. I chose to simply wait for a connection, as the US Robotics modem has a built-in speaker and I won't be putting this S-7600A/PIC16F877 Internet Engine out in the real world. I arbitrarily set the wait time for 20 s, which should be long enough for a connection to be established at 2400 bps. You should write some code to track modem and line status if you plan to deploy your S-7600A/PIC16F877 Internet Engine in the field.

GO

After the PIC16F877 is notified that the physical connection to the ISP is established, the PIC is responsible for passing the ball to the S-7600A. This is done by setting the Connection Valid bit in the S-7600A PPP control and status register. Setting this bit tells the network stack that the layer below it is up and operational. The Use PAP and PPP enabled bits also reside in the PPP control and status register and are set with the same command used to signal

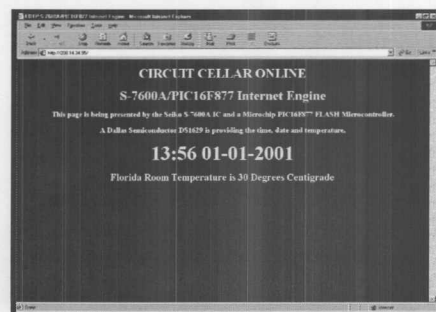


Photo 9—Here's the light at the end of the Internet tunnel!

a good connection to the ISP. If you follow WWII movies, this is where the pilot of the B-17 turns over control to the guy with the Norden bombsight. The PIC sets the SCTL bit in the serial port configuration register and gives control of the S-7600A UART to the Seiko IC and its network stack mechanism.


This is the point in the process where the S-7600A performs its magic. I've taken a full snapshot of the process using Serialtest Async. I'm providing it to you as a readable file so you can follow through the PPP negotiation sequence frame by frame, beginning to end. The only information I'll censor is my logon password. I'm also including an Acrobat file that contains the Serialtest Async frame decodes. Using the ASCII file in conjunction with the PDF file will illustrate the total PPP process.

Bit 0 of the PPP control and status register confirms to the PIC16F877 that PPP is up and operating. Your ISP assigned IP address is negotiated and Photo 8 is the Tera Term Pro view of the result.

The S-7600A/PIC16F877 Internet Engine's PIC16F877 code has now entered the web server loop area. The code here checks PPP up status and the condition of the modem's DCD pin to determine if the link is up and functioning. The S-7600A is capable of supporting two sockets. This web server application uses Socket 0. Because the S-7600A/PIC16F877 Internet Engine is serving, Port 80 is loaded into the S-7600A, the server mode is activated, and Socket 0 is brought online.

If no timeouts occur and the link remains active, a request from a remote web browser can be fielded. S-7600A socket status is interrogated to ensure that a good connection exists and that the TCP state is listening. If a request is received from a web browser, an HTTP header is constructed and transmitted followed by the HTML stored in the 24LC256 EEPROM. Time and temperature are obtained from the DS1629 and inserted into the HTML text. The assembled HTML page is then sent to the S-7600A's socket data register. The data is sent and the PIC16F877 waits

for the transmit to complete. The socket is then closed and reopened making it ready to serve yet another page of HTML like the one shown in Photo 9.

I've shown you how to take an everyday PIC, a C compiler, a tiny firmware protocol stack, and a handful of common components and put them on the Internet. For those of you who wish to build and experiment with your own S-7600A/PIC16F877 Internet Engine, I'll post the details of how to purchase the kit at www.edtp.com. 

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

Thanks to Mike Garbutt, Stephen Humberd, and Rodger Richey for the excellent PIC application notes (Microchip AN731 and AN732) that I used as models for the S-7600A/PIC16F877 Internet Engine.

SOURCES

SP3243E

Sipex Corp.

(978) 667-8700

Fax: (978) 670-9001

www.sipex.com

PIC16F877 and 24LC256

Microchip Technology, Inc.

(888) 628-6247

(480) 786-7200

Fax: (480) 899-9210

www.microchip.com

DS1629

Dallas Semiconductor

(972) 371-4448

Fax: (972) 371-3715

www.dalsemi.com

LM2937

National Semiconductor

(408) 721-5000

Fax: (408) 739-9803

www.national.com

CH1786ET

Cermetek

(408) 752-5000

(800) 882-6271

Fax: (408) 752-5004

www.cermetek.com

Serialtest Async

Frontline Test Systems

(804) 984-4500

Fax: (804) 984-4505

www.fte.com

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.

Listing 1—

```

/*****
 * BOOT LOADER CODE FOR THE PIC16F877
 * CIRCUIT CELLAR ONLINE VERSION
 * CODE LAST UPDATED 12/31/2000
 * WRITTEN BY: FRED EADY
 *****/

#include <16F877.H>
#include <f877.H>
#fuses HS,NOWDT,NOPROTECT,NOBROWNOUT,NOLVP,PUT
#id 0X1231

/*****
 * DEFINE END OF USER CODE SPACE
 *****/

#define MAXADDR 0x1D00

/*****
 * DEFINE INTEL HEX BUFFER AREA OF 64 BYTES IN BANK 2 *
 *****/

#define BUFFER_LEN 64
int buffaddr;
#byte buffaddr = 0xA0
char buffer[BUFFER_LEN];
#byte buffer = 0xA1

/*****
 * DEFINE FLOW CONTROL PINS
 *****/

#define CTSON output_low(PIN_B2)
#define CTSOFF output_high(PIN_B2)
#define RTS input(PIN_B1)

/*****
 * TELL COMPILER NOT TO PLACE THIS CODE INLINE
 *****/

#SEPARATE
unsigned int atoi_b16(char *s);

/*****
 * INLINE RECEIVE CHARACTER ROUTINE
 *****/

#inline
char rxchar(){
char serial_in;
#asm
    norx:
    btfss RCIF
    goto norx
    movf RCREG,W
    movwf serial_in
#endasm
return(serial_in);
}

/*****
 * INLINE TRANSMIT CHARACTER ROUTINE
 *****/

#inline
void txchar(char s){
char serial_out;
serial_out=s;
#asm
    notx:
    btfss TXIF
    goto notx
    movf serial_out,W
    movwf TXREG
#endasm
}

/*****

```

(continued)

Listing 1—continued

```

 * ASCII TO INTEGER CONVERSION ROUTINE
 *****/

#ORG 0x1D80,0x1DCF
unsigned int atoi_b16(char *s){
unsigned int result = 0;
int i;

for (i=0; i<2; i++) {
    if (*s >= 'A' && *s <= 'F')
        result = 16*result + (*s) - 'A' + 10;
    else if (*s >= '0' && *s <= '9')
        result = 16*result + (*s) - '0';

    s++;
}

return(result);
}

/*****
 * MAIN BOOT LOADER ROUTINE
 *****/

#ORG 0x1D00,0x1FFF
void main(void)
{
    char byte_in;
    short code_good, not_done, no_char, bootcodeflag;
    int i, count, line_type, cs, check_sum, addr1;
    long
bootaddr, data, addr, addrh, bootcodeaddr, timr1, timrh;

/*****
 * GET BOOTCODE VECTOR ADDRESS
 *****/

#asm
    movphw    bootcode
    movwf     addrh
    movplw    bootcode
    movwf     addr1
#endasm
bootcodeaddr = addrh << 8 | addr1;

/*****
 * INITIALIZE THE PIC UART 2400/N/8/1
 * RB1 = RTS    RB2 = CTS
 *****/

TRISB = 0XFB;
CTSOFF;
SPBRG = 0X2F;
TXSTA = 0x22;
SPEN = 1;
CREN=1;

/*****
 * WAIT FOR UPLOAD COMMAND FROM TERA TERM
 * IF NO COMMAND RECEIVED DURING TIMEOUT, RUN BOOTCODE *
 * u = download new code
 *****/

timr1=0x20; // DELAY VALUE
timrh=0xFFFF; // DELAY VALUE
no_char = TRUE;
CTSON;

do{

    if(bit_test(PIR1,5)){
        byte_in = rxchar();
        no_char = FALSE;
    }
    --timrh;

    if(timrh == 0x00){

```

(continued)

Listing1—(continued)

```

        txchar('*');
        -timr1;
        timrh = 0xFFFF;
    }
}while ((timr1 != 0x00) && (no_char == TRUE));

if(timr1 != 0x00 && no_char == FALSE && byte_in ==
'u'){
    CTSOFF;
    goto download;
}

/*****
* BOOTCODE VECTOR
*****/

    #asm
    clr f PCLATH
bootcode:
    nop
    nop
    nop
    nop
    goto 0x00
    #endasm

/*****
* DOWNLOAD INTEL HEX FILE FROM TERA TERM PRO
*****/

download:

    //not_done = TRUE;
    CTS0N;

    while (1){
        buffaddr = 0;
        code_good = TRUE;

        do {
            buffer[buffaddr++] = rxchar();

        } while ((buffer[buffaddr-1] != 0x0D) &&
(buffaddr <= BUFFER_LEN));

        CTSOFF;

/*****
* INTEL HEX PROCESSOR
* : = START OF HEX LINE
* AA = NUMBER OF BYTES IN THE ASCII LINE
* BBCC = LINE ADDRESS
* DD = LINE TYPE
*****/

/*****
* PROCESS THE INTEL HEX LINE HEADER
*****/

        //:
        if (buffer[0] == ':') {
            //:AA
            count = atoi_b16 (&buffer[1]);
            //:AABB
            addr = atoi_b16 (&buffer[3]);
            //:AABBC
            addr = (addr << 8) | atoi_b16 (&buffer[5]);
            //:AABBCDD
            line_type = atoi_b16 (&buffer[7]);
/*****
* ADJUST THE ADDRESS
*****/

            if (addr != 0)
                addr /= 2;

/*****
* END OF HEX FILE DATA IS LINE TYPE = 0X01
*****/

```

(continued)

Listing1—(continued)

```

*****/

        if (line_type == 1)
            break;

/*****
* COMPUTE THE HEX LINE CHECKSUM
*****/

        else {
            cs = 0;
            for (i=1; i<(buffaddr-3); i+=2)
                cs += atoi_b16 (&buffer[i]);

            cs = 0xFF - cs + 1;
            check_sum = atoi_b16 (&buffer[buffaddr-3]);

            if (check_sum != cs)
                code_good = FALSE;

/*****
* PROCESS THE INTEL HEX DATA AREA
*****/

            else {
                i = 9;
                while (i < buffaddr-3) {
                    data = atoi_b16 (&buffer[i+2]);
                    data = (data << 8) | atoi_b16 (&buffer[i]);

/*****
* PLACE BOOT VECTOR IN BOOTCODE AREA IF addr < 0X0004
*****/

                    bootcodeflag = FALSE;
                    if(addr < 0x0004){
                        bootcodeflag = TRUE;
                        bootaddr = addr + bootcodeaddr;
                        write_program_eeeprom (bootaddr,data);
                    }

/*****
* WRITE THE INTEL HEX DATA TO PROGRAM MEMORY
*****/

                    if((bootcodeflag == FALSE) && (addr < MAXADDR)){
                        write_program_eeeprom (addr,data);
                        ++addr;
                        i += 4;
                    }
                }
            }

            CTS0N;
        }

/*****
* IF INTEL HEX DOWNLOAD IS SUCCESSFUL
* SEND g TO TERA TERM AND RESTART
*****/

        CREN = 0;
        timrh = 0xFFFF;
        timr1 = 0x40;
        CTS0N;

        do{

            -timrh;

            if(timrh == 0x00){
                txchar('*');
                -timr1;
                timrh = 0xFFFF;
            }
        }while (timr1 != 0x00);

        if (code_good)
            txchar('g');

```

(continued)

Listing 1—continued

```

/*****
* IF INTEL HEX DOWNLOAD IS IN ERROR
* PUT NOPs AT BOOTCODE VECTOR AREA
* SEND e TO TERA TERM AND RESTART
*****/

    else{
        addr = 0x00;
        data = 0x00;
        while (addr < 4){
            bootaddr = addr + bootcodeaddr;
            write_program_eeprom (bootaddr,data);
            ++addr;
        }
        txchar('E');
    }
/*****
* ABSOLUTE RESET VECTOR TO LOCATION 0X0000
*****/
#asm
    MOVLW 0x00
    MOVWF 0x0A
    GOTO 0x00
#endasm
}

```

Listing 2 —

```

; CIRCUIT CELLAR ONLINE
; SEIKO S-7600A/PIC16F877 INTERNET ENGINE UPLOAD MACRO
; 01/01/2001
; WRITTEN BY: FRED EADY

; DEFINITIONS

boxtitle = 'Circuit Cellar Online'
uploadprompt = 'Do you wish to upload new PIC16F877 code?'
uploadfile = 'seiko.hex'
uploadgood = 'Upload Complete..'
uploadbad = 'Upload Error..'
linkdown = 'Link is not established..'
linkup = 'Link established with remote host..'
hostdown = 'Unable to communicate with remote host..'

; MAIN MACRO CODE

:begin
connect '/C=2'
if result = 0 goto nolink
if result = 1 goto nohost
if result = 2 goto goodlink

:nolink
messagebox linkdown boxtitle
disconnect
end

:nohost
messagebox hostdown boxtitle
disconnect
end

:goodlink
yesno box uploadprompt boxtitle
if result goto loadcode
goto noload

:loadcode
send 'u'
sendfile uploadfile 0
wait 'g' 'E'

```

(continued)

Listing 2—continued

```

if result = 1 goto donegood
if result = 2 goto uploadererror

:donegood
messagebox uploadgood boxtitle
end

:noload
end

:uploadererror
messagebox uploadbad boxtitle
end

```

Listing 3 —

```

<TITLE>EDTP S-7600A/PIC16F877 Internet Engine</TITLE>
<body bgcolor=#000000 ><CENTER><font size="6"
color="#FFFFFF"><b>CIRCUIT CELLAR ONLINE</b></font>
<p><font size="6" color="#00FF00"><b>S-7600A/PIC16F877
Internet Engine</b></font>
<p><font size="4" color="#FFFFFF"><b>This page is being
presented by the Seiko S-7600A IC and a Microchip
PIC16F877 FLASH Microcontroller.</font>
<p><font size="4" color="#FFFFFF">A Dallas Semiconductor
DS1629 is providing the time, date and temperature.</
font></p>
<p><font size="8" color="#F0F0FF">%t</font></p>
<p><font size="5" color="#FFFFFF">Florida Room Tempera-
ture is %c Degrees Centigrade</font></p></CENTER>;

```

Listing 4—

```

#include "16f877.h"
#include "f877.h"
#include "s7600.h"
#include <ctype.h>
#include <string.h>

#fuses HS,NOWDT,NOPROTECT,NOBROWNOUT,NOLVP

#define EEPROM_SDA PIN_C4
#define EEPROM_SCL PIN_C3
#define hi(x) ((*(&x+1))

#use delay(clock=7372800)
#use rs232(baud=2400, xmit=PIN_C6, rcv=PIN_C7)
#use standard_io(A)
#use standard_io(B)
#use standard_io(C)
#use standard_io(E)
#use i2c(master,sda=EEPROM_SDA, scl=EEPROM_SCL)

#define EEPROM_ADDRESS long int
#define EEPROM_SIZE 1024
#define esc 0x1B

// PORTA bits
#bit READX = PORTA.1
#bit CS = PORTA.2
#bit RS = PORTA.3

// PORTB bits
#bit CTS = PORTB.2
// When CTS = 0 send is enabled
#bit BUSY = PORTB.3
#bit INT1 = PORTB.4

```

(continued)

Listing 4—continued

```
#bit WRITEX = PORTB.5
// PORTC bits
#bit RESET = PORTC.0

short negtemp;
char i,j,ch,addr,temp,pot,ftmp,ctmp,count,count1,page;
long index,byt_cnt,hits;
char Login[10];
char MyIPAddr[4];
char user[33];
char pass[33];
char phone[17];
char read_data[5];
char test_str[7];
char read_Q[7];
char mins[3];
char hours[3];
char day[2];
char date[3];
char month[3];
char year[3];

#include "seiko_ct.c" //
Seiko routines use code from AN732

/*****
** char DataAvailable(void)
** Determines if there is any data available to read out of
** the S-7600A.
** Returns the value of the data available bit from the
** S-7600A.
*****/
char DataAvailable(void)
{
    return (ReadSeiko(Serial_Port_Config)&0x80);
}

/*****
** void S_Putc(char data)
** Writes a byte of data to the serial port on the S-7600A.
*****/
void S_Putc(char data)
{
    while(!BUSY);
    CS = 1;
    RS = 0;
    WRITEX = 0;
    PORTD = Serial_Port_Data;
    TRISD = 0;
    READX = 1;
    READX = 0;
    WRITEX = 1;
    RS = 1;
    CS = 0;

    CS = 1;
    WRITEX = 0;
    PORTD = data;
    READX = 1;
    READX = 0;
    WRITEX = 1;
    CS = 0;

    TRISD = 0xff;
}

/*****
** void W_Putc(char data)
** Writes a byte of data to the socket on the S-7600A.
*****/
void W_Putc(char data)
{
    // Make sure that the socket buffer is not full
    while(0x20==(ReadSeiko(0x22)&0x20))
    {
        WriteSeiko(TCP_Data_Send,0);
        while(ReadSeiko(Socket_Status_H));
    }
}
```

(continued)

Listing 4—continued

```
while(!BUSY);
CS = 1;
RS = 0;
WRITEX = 0;
PORTD = Socket_Data;
TRISD = 0;
READX = 1;
READX = 0;
WRITEX = 1;
RS = 1;
CS = 0;

CS = 1;
WRITEX = 0;
PORTD = data;
READX = 1;
READX = 0;
WRITEX = 1;
CS = 0;

TRISD = 0xff;
}

#include "eeprom.c" // external serial
EEPROM routines from AN732

/
*****/
void Get_username(void)
{
    // Requests and reads the user name from the input terminal.
    *****/
void Get_username(void)
{
    char n_tmp;
    i=0;
    printf("%c[2J",esc);
    printf("%c[12;20H 32 chars max",esc);
    printf("%c[10;20H Enter user name: ",esc);

    while(1)
    {
        user[i]=0;
        ch=GETC();
        if(ch==0x0D)
            break;
        putc(ch);
        if(ch != 0x08)
        {
            user[i]=ch;
            i++;
        }
        if(i==32) break;
    }

    // write user name to the EEPROM
    n_tmp=0;
    for(i=0;i<0x1F;i++)
    {
        ch=user[i];
        write_ext_eeprom(n_tmp,user[i]);
        n_tmp++;
    }
}

/
*****/
void Get_password(void)
{
    // Requests and reads the password from the input terminal.
    *****/
void Get_password()
{
    byte a_tmp;
    i=0;
    printf("%c[2J",esc);
}
```

(continued)

Listing 4—(continued)

```

printf("%c[12;20H 32 chars max",esc);
printf("%c[10;20H Enter password: ",esc);

while(1)
{
    pass[i]=0;
    ch=getc();
    if(ch==0x0D)
        break;
    if(ch != 0x0A)
    {
        putc(ch);
        if(ch != 0x08)
        {
            pass[i]=ch;
            i++;
        }
    }
    if(i==16) break;
}

// write password to the EEPROM
a_tmp=0x20;
for(i=0;i<0x1F;i++)
{
    ch=pass[i];
    write_ext_eeprom(a_tmp, pass[i]);
    a_tmp++;
}
}

/
*****
** void Get_phone(void)
**
** Requests and reads the telephone number from the input
** terminal.
**
*****/
void Get_phone()
{
    byte p_tmp;
    printf("%c[2J",esc);
    printf("%c[12;20H 16 chars max",esc);
    printf("%c[10;20H Enter phone number: ",esc);

    i=0;
    while(1)
    {
        phone[i]=0;
        ch=getc();
        if(ch==0x0D)
            break;
        if(ch != 0x0A)
        {
            putc(ch);
            if(ch != 0x08)
            {
                phone[i]=ch;
                i++;
            }
        }
        if(i==16) break;
    }

    // write phone number to the EEPROM
    p_tmp=0x40;
    for(i=0;i<16;i++)
    {
        ch=phone[i];
        write_ext_eeprom(p_tmp, phone[i]);
        p_tmp++;
    }
}

/
*****
** void Get_Time(void)
**

```

(Continued)

Listing 4—(continued)

```

** Resets time and date
**
**
*****/
void Get_Time(void){
tryminsagain:
    printf("%c[2J",esc);
    printf("%c[10;20H Enter minutes 00-59: ",esc);

    i=0;
    while(1)
    {
        mins[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
        {
            mins[i]=ch;
            i++;
        }
    }
    else
        goto tryminsagain;

    if(i==2)
        break;
}

if(mins[0] > 0x35)
    goto tryminsagain;

swap(mins[0]);
mins[0] &= 0xF0;
mins[1] &= 0x0F;
mins[2] = mins[0] | mins[1];

tryhoursagain:
    printf("%c[11;20H Enter hours 00-23: ",esc);

    i=0;
    while(1)
    {
        hours[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
        {
            hours[i]=ch;
            i++;
        }
    }
    else
        goto tryhoursagain;

    if(i==2)
        break;
}

if(hours[0] == 0x32 && hours[1] > 0x33)
    goto tryhoursagain;
if(hours[0] > 0x32)
    goto tryhoursagain;
swap(hours[0]);
hours[0] &= 0xF0;
hours[1] &= 0x0F;
hours[2] = hours[0] | hours[1];

trydayagain:
    printf("%c[12;20H Enter day 1-7 (Sunday = 1): ",esc);

    i=0;
    while(1)
    {
        day[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
            day[i]=ch;
        else

```

(Continued)

Listing 4—continued

```

        goto trydayagain;
        break;
    }
    if(day[0] > 0x37)
        goto trydayagain;
    day[0] = day[0] & 0x0F;

trydateagain:
    printf("%c[13;20H Enter Date 01-31: ",esc);

    i=0;
    while(1)
    {
        date[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
        {
            date[i]=ch;
            i++;
        }
    }
    else
        goto trydateagain;

    if (i==2)
        break;
    }
    if(date[0] > 0x33)
        goto trydateagain;
    if(date[0] == 0x00 && date[1] == 0x00)
        goto trydateagain;
    if(date[0] == 0x33 && date[1] > 0x31)
        goto trydateagain;

swap(date[0]);
date[0] &= 0xF0;
date[1] &= 0x0F;
date[2] = date[0] | date[1];

trymonthagain:
    printf("%c[14;20H Enter Month 01-12: ",esc);

    i=0;
    while(1)
    {
        month[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
        {
            month[i]=ch;
            i++;
        }
    }
    else
        goto trymonthagain;

    if (i==2)
        break;
    }
    if(month[0] > 0x31)
        goto trymonthagain;
    if(month[0] == 0x00 && month[1] == 0x00)
        goto trymonthagain;
    if(month[0] == 0x31 && month[1] > 0x32)
        goto trymonthagain;

swap(month[0]);
month[0] &= 0xF0;
month[1] &= 0x0F;
month[2] = month[0] | month[1];

tryyearagain:
    printf("%c[15;20H Enter Year 00-99: ",esc);

    i=0;
    while(1)
    {

```

(continued)

Listing 4—continued

```

        year[i]=0;
        ch=getc();

        putc(ch);
        if(ch < 0x3A && ch > 0x2F)
        {
            year[i]=ch;
            i++;
        }
    }
    else
        goto tryyearagain;

    if (i==2)
        break;
    }

swap(year[0]);
year[0] &= 0xF0;
year[1] &= 0x0F;
year[2] = year[0] | year[1];

i2c_start();
i2c_write(0x9E);
i2c_write(0xC0);
i2c_write(0x00);
i2c_write(0x00); // set seconds to 00
i2c_write(mins[2]); // set minutes
i2c_write(hours[2]); // set hours
i2c_write(day[0]); // set day
i2c_write(date[2]); // set date
i2c_write(month[2]); // set month
i2c_write(year[2]); // set year
i2c_stop();

}

/
*****
** void Read_file(void)
**
** Requests and reads the HTML web page that is sent when
** requested by a web browser.
**
** This routine strips out all carriage returns and line-
** feeds found in the file. It also looks for a semi-
** colon to end the file.
**
** *****/
void Read_file()
{
    printf("%c[2J",esc); // Print request to the terminal
    printf("%c[10;20H Ready to receive file",esc);
    printf("%c[12;20H 32688 bytes max",esc);
    printf("%c[14;20H End file with ;",esc);
    printf("%c[16;20H Set your terminal for
        Hardware Flow Control",esc);
    ch=1;
    i=0;
    index=0x50;

    while(1)
    {
        CTS = 0;
        ch=getc();
        CTS = 1;
        if(index == 32767)
            break;
        if(ch==00)
            break;
        if(ch==';')
            break;

        // Otherwise write character to EEPROM
        write_ext_eeprom(index, ch);

        index++;
    }

```

(continued)

Listing 4—continued

```

}
write_ext_eeprom(index, 0); // Write terminating NULL
    CTS = 0;

    // Print status of download to EEPROM
    index = index - 80;
    printf("%c[2J", esc);
    printf("%c[12;28H Received %lu bytes", esc, index);
    if(index > 32688)
printf("%c[18;20H Error maximum bytes is 32688", esc);
    printf("%c[18;25H Press any key to continue", esc);
    ch=getc();
    CTS = 1;
}

/
*****
** void init_temp
**
** initialize the DS1629
**
**
**
**
void init_temp(){
    i2c_start();
    i2c_write(0x9E);
    i2c_write(0xAC);
    i2c_write(0x00);
    i2c_stop();
}

/
*****
** void Menu(void)
**
** Displays menu on user's terminal screen. Allows changes
** to username, password, phone number and web page.
**
**
**
void Menu(void)
{
    i=0;

    while(ch != 0x1B)
    {
        CTS=0;

        printf("%c[2J", esc);
        printf("%c[6;25H 1 Enter user name", esc);
        printf("%c[8;25H 2 Enter password", esc);
        printf("%c[10;25H 3 Enter phone number", esc);
        printf("%c[12;25H 4 Set Time and Date", esc);
        printf("%c[14;25H 5 Down load HTML file", esc);
        printf("%c[17;30H ESC exit", esc);

        ch=getc(); // Get input and process
        switch(ch)
        {
            case 0x31: // '1' -> change username
                Get_username();
                break;

            case 0x32: // '2' -> change password
                Get_password();
                break;

            case 0x33: // '3' -> change phone #
                Get_phone();
                break;

            case 0x34: // '4' -> change time and date
                Get_Time();
                break;

            case 0x35: // '5' -> new web page
                Read_file();
                break;
        }
    }
}

```

(continued)

Listing 4—continued

```

}
    CTS=1;
}

/
*****
** void main(void)
**
**
**
void main(void)
{
    // Initialize PORTs & TRISs
    //BIT 1 READX
    // 2 CS
    // 3 RS
    PORTA = 0xF9; //11111001

    //BIT 2 CTS
    // 3 BUSY
    // 4 INT1
    // 5 WRTX
    PORTB = 0x38; //00111000

    //BIT 0 RESET
    PORTC = 0x80; //10000000

    PORTD = 0x00; //00000000

    //BIT 0 LED1
    // 1 LED2
    PORTE = 0xFA; //11111010

    TRISA = 0xD1; //11010001
    TRISB = 0x18; //00011000
    TRISC = 0x80; //10000000
    TRISD = 0xFF; //11111111
    TRISE = 0x00; //00000000

    ADCON1 = 0x06; //00000110 all digital
    ADCON0 = 0;

    T1CON = 0x31; //00110001 Timer1

    CTS = 1;

    init_ext_eeprom();
    init_temp();

    Menu();

restart:
    RESET = 0;
    WriteSeiko(Serial_Port_Config, 0x06);
    delay_ms(10);
    RESET = 1;

    WriteSeiko(Clock_Div_L, 0xE5);
    while(ReadSeiko(Clock_Div_L) != 0xE5)
        WriteSeiko(Clock_Div_L, 0xE5);
    WriteSeiko(Clock_Div_H, 0x00);

    WriteSeiko(BAUD_Rate_Div_L, 0x5F);
    WriteSeiko(BAUD_Rate_Div_H, 0x00);

    WriteSeiko(Serial_Port_Config, 0x00);

    printf(S_Putc, "AT&F\r");
    delay_ms(10);

    WriteSeiko(PPP_Control_Status, 0x01);
    WriteSeiko(PPP_Control_Status, 0x00);
    WriteSeiko(PPP_Control_Status, 0x20);
    delay_ms(5);
}

```

(continued)

(Continued) —

```
        ch=1;
        i=0;
        while(ch)
        {
            ch = read_ext_eeprom(i);
            i++;
        }
        i--;
        WriteSeiko(PAP_String,i);

        for(j=0; j<i; j++)
        {
            ch = read_ext_eeprom(j);
            WriteSeiko(PAP_String,ch);
        }

        ch=1;
        i=0x20;
        while(ch)
        {
            ch = read_ext_eeprom(i);
            i++;
        }
        i--;
        i=(i-0x20);
        WriteSeiko(PAP_String,i);

        for(j=0x20; j<(i + 0x20); j++)
        {
            ch = read_ext_eeprom(j);
            WriteSeiko(PAP_String,ch);
        }

        WriteSeiko(PAP_String,0x00);

        printf(S_Putc,"ATDT");
        ch=1;
        index=0x40;
        while(1)
        {
            ch = read_ext_eeprom(index);
            if(ch == 0)
                break;

            S_Putc(ch);
            index++;
        }

        printf(S_Putc,"\r");

        delay_ms(5);

        printf("%c[2J",esc);
        printf("\rDialing");

        ch=1;
        i=0x40;
        while(1)
        {
            ch = read_ext_eeprom(i);
            if(ch == 0)
                break;
            printf("%c",ch);
            i++;
        }
        printf("\r");

        delay_ms(20000);

        WriteSeiko(PPP_Control_Status,0x62);
        WriteSeiko(Serial_Port_Config,0x01);
        delay_ms(5);

        while(!(ReadSeiko(PPP_Control_Status)&0x01))
            delay_ms(5);
```

(continued)

(Continued) —

```
        while(ReadSeiko(Our_IP_Address_L) == 0);

        MyIPAddr[0] = ReadSeiko(Our_IP_Address_L);
        MyIPAddr[1] = ReadSeiko(Our_IP_Address_M);
        MyIPAddr[2] = ReadSeiko(Our_IP_Address_H);
        MyIPAddr[3] = ReadSeiko(Our_IP_Address_U);

        printf("\r\nMy address is
        %u.%u.%u.%u",MyIPAddr[3],MyIPAddr[2],MyIPAddr[1],MyIPAddr[0]);

        while(1)
        {
            while(1)
            {
                delay_ms(1);

                if(!(ReadSeiko(PPP_Control_Status)&0x01))
                    goto restart;

                if(ReadSeiko(Serial_Port_Config)&0x40)
                    goto restart;

                WriteSeiko(Socket_Index,0x00);
                WriteSeiko(Socket_Config_Status_L,0x10);
                delay_ms(10);

                WriteSeiko(Our_Port_L,80);
                WriteSeiko(Our_Port_H,0);

                WriteSeiko(Socket_Config_Status_L,0x06);

                WriteSeiko(Socket_Activate,0x01);
                delay_ms(5);

                printf("\n\rSocket open\n\r");
                i = 2;
                while(1)
                {
                    delay_ms(1);

                    if(!(ReadSeiko(PPP_Control_Status)&0x01))
                        goto restart;

                    if(ReadSeiko(Serial_Port_Config)&0x40)
                        goto restart;

                    temp =
                    ReadSeiko(Socket_Status_M);
                    if(temp&0x10)
                    {
                        i = 0;

                        break;
                    }
                    else if(temp&0xe0)
                    {
                        break;

                        if(temp == 0x09)
                            continue;

                        delay_ms(5);

                        i++;
```

(continued)

Listing 4—continued

```

        if(i == 255)                break;
    }
    if(!i)                          break;
}

if(i == 1)                          break;

printf("\n\rWaiting for data");
WriteSeiko(Socket_Interrupt_H,0xf0);
i=0;

printf("\n\rReading data\r\n");

while(ReadSeiko(Socket_Config_Status_L)&0x10)
{
    temp = ReadSeiko(Socket_Data);
    putc(temp);
}

WriteSeiko(Socket_Data,0x0A);
WriteSeiko(Socket_Data,0x0D);
WriteSeiko(Socket_Data,0x0A);
WriteSeiko(Socket_Data,0x0D);

byt_cnt=0;
index=0x50;
ch=1;
while(ch != 0)
{
    ch =
    read_ext_eeprom(index);

    if(ch == 0)                      break;
    if(ch == 0x25)
    {
        index++;
        ch =
        read_ext_eeprom(index);

        switch(ch)
        {
            case 'a':

printf(W_putc,"%u.%u.%u.%u",MyIPAddr[3],MyIPAddr[2],
MyIPAddr[1],MyIPAddr[0]);

                i2c_start();
                i2c_write(0x9E);
                i2c_write(0xEE);

                delay_ms(500);

                i2c_start();
                i2c_write(0x9E);
                i2c_write(0xAA);
                i2c_start();
                i2c_write(0x9F);
                ctmp = i2c_read();
                i2c_stop();

                negtemp = FALSE;
                if(ctmp >= 0x80){
                    ctmp = !ctmp +1;
                    negtemp = TRUE;
                }

                if(negtemp==TRUE)
                    printf(W_putc,"-");

printf(W_putc,"%u",ctmp);

                break;
            case 't':
                i2c_start();
                i2c_write(0x9E);

```

(continued)

Listing 4—continued

```

i2c_write(0xC0);
i2c_write(0x01);

i2c_start();
i2c_write(0x9F);
//i = i2c_read();
mins[0] = i2c_read();
hours[0] = i2c_read();
day[0] = i2c_read();
date[0] = i2c_read();
month[0] = i2c_read();
year[0] = i2c_read();
i2c_stop();

negtemp = FALSE;
if(ctmp >= 0x80){
    ctmp = !ctmp +1;
    negtemp = TRUE;
}

//secs[2] = secs[0];
//secs[0] &= 0xF0;
//secs[0] >>= 4;
//secs[0] &= 0x0F;
//secs[0] += 0x30;
//secs[1] = secs[2];
//secs[1] &= 0x0F;
//secs[1] += 0x30;

mins[2] = mins[0];
mins[0] &= 0xF0;
mins[0] >>= 4;
mins[0] &= 0x0F;
mins[0] += 0x30;
mins[1] = mins[2];
mins[1] &= 0x0F;
mins[1] += 0x30;

hours[2] = hours[0];
hours[0] &= 0xF0;
hours[0] >>= 4;
hours[0] &= 0x0F;
hours[0] += 0x30;
hours[1] = hours[2];
hours[1] &= 0x0F;
hours[1] += 0x30;

//day[0] &= 0x0F;
//day[0] += 0x30;

date[2] = date[0];
date[0] &= 0xF0;
date[0] >>= 4;
date[0] &= 0x0F;
date[0] += 0x30;
date[1] = date[2];
date[1] &= 0x0F;
date[1] += 0x30;

month[2] = month[0];
month[0] &= 0xF0;
month[0] >>= 4;
month[0] &= 0x0F;
month[0] += 0x30;
month[1] = month[2];
month[1] &= 0x0F;
month[1] += 0x30;

year[2] = year[0];
year[0] &= 0xF0;
year[0] >>= 4;
year[0] &= 0x0F;
year[0] += 0x30;
year[1] = year[2];
year[1] &= 0x0F;
year[1] += 0x30;

printf(W_putc,"%c%c%c%c%c%c%c%c-
20%c%c",hours[0],hours[1],

```

(continued)

Listing 4—continued

```

mins[0],mins[1],month[0],month[1],date[0],date[1],year[0],year[1]);
    break;
    }
    else
    {
        count=0;
        count1=0;

        bit_clear(PIR1,TMR1IF);
        while(0x20==(ReadSeiko(0x22)&0x20))
        {

            WriteSeiko(TCP_Data_Send,0);
            while(ReadSeiko(Socket_Status_H));

            if(ReadSeiko(Serial_Port_Config)&0x40)
            goto restart;
            if(bit_test(PIR1,TMR1IF))
            {

                count++;
                if(count > 0xf8)
                {
                    count1++;
                    count=0;
                }

                if(count1 > 0xc0)
                goto restart;
                bit_clear(PIR1,TMR1IF);
            }

            WriteSeiko(Socket_Data,ch);
            }
            index++;

            WriteSeiko(TCP_Data_Send,0);

            count=0;
            bit_clear(PIR1,TMR1IF);
            while(0x40!=(ReadSeiko(0x22) & 0x40))
            {

                if(bit_test(PIR1,TMR1IF))
                {
                    count++;
                }

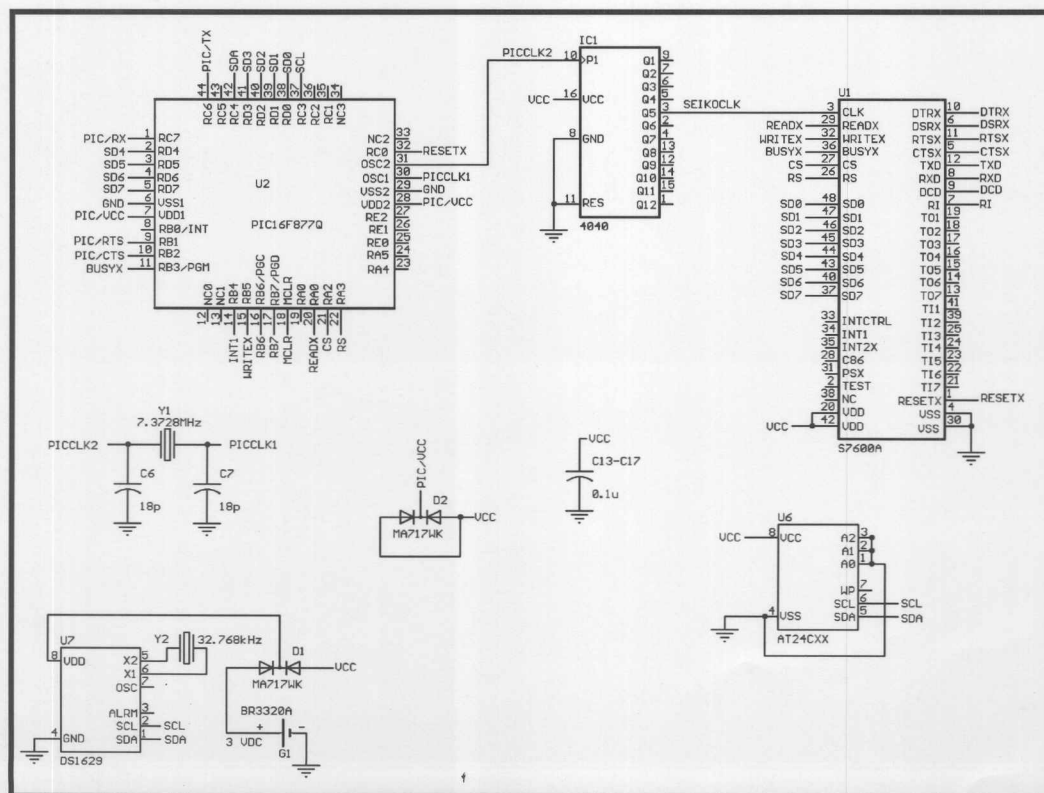
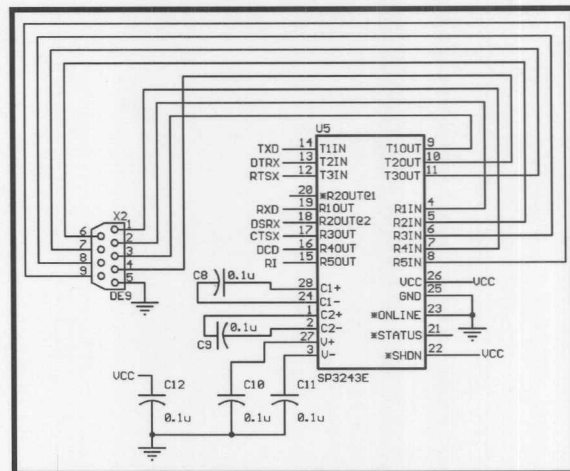
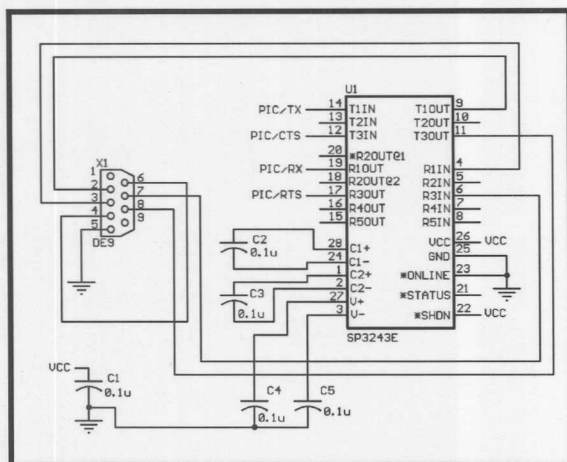
                bit_clear(PIR1,TMR1IF);

                printf("close socket\n");
                WriteSeiko(Socket_Activate,0);
                WriteSeiko(TCP_Data_Send,0);

                for(i=0;i<255;i++)
                {
                    delay_ms(10);
                    temp = ReadSeiko(Socket_Status_M);
                    if((temp & 0x0f)==0x08)
                    break;
                    if((temp&0xe0))
                    break;
                }
                printf("\n\rfinal socket wait\n");
            }
            while(ReadSeiko(Socket_Status_H));
            delay_ms(5000);
        }
    }
}

```

(continued)



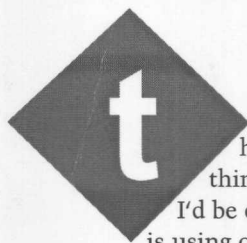
FEATURE ARTICLE

Fred Eady

An S-7800A/PIC16F877 Journey

Part 3: Hot-Wiring the System

Using a PIC to communicate over the Internet? Fred is just as surprised as you, but nevertheless, that is the task at hand for this month. With in-circuit programmable flash memory and four times the I/O space of the old PIC16C55, he started the ball rolling with SMTP. Once you know how to send e-mail with small embedded devices, you've hit guru status.



he absolute last thing I ever thought I'd be doing with a PIC is using one to communi-

cate over the Internet. Way back in the early '80s, I was enthralled with the PIC16C55. What more could an embedded engineer want or need? This puppy had lots of I/O pins and program memory galore! Shucks! I could turn on LEDs and flip relays all day with a minimal amount of thought or code. One of the neatest tricks was making a virtual UART so the PIC could talk to the 20-MHz desktop's serial port running Visual Basic 3 on Win3.1.

Times have truly changed. When I designed my first PIC programmer for the PIC16C5x parts, the Microchip assembler was not a freebie. In fact, I still remember the Microchip rep calling me in the middle of the night with the price for the assembler. It was a whopping \$99.95 and came on a 5.25" floppy. Immediately I replied, "How am I going to sell that with my programmer kit when the PIC programmer hardware only costs \$69?" No problem. I hung up the phone, got out of bed, and started writing my own PIC assembler. The rest is still attempting to make history.

Today, the Microchip MPLAB suite is free for downloading, and the new PIC parts like the PIC16F877 on the S-7600A/PIC16F877 Internet Engine are sporting in-circuit programmable flash memory with four times the program memory and I/O space of the lowly but popular PIC16C55. I was running a BBS then, as the Internet had not yet become a household item. Now, the Internet is the driving force behind much of today's technology. PIC compilers are abundant and PIC programmers are everywhere. So, there's nothing to keep an embedded designer from putting his or her device on the 'Net. In fact, the PIC tools have become sophisticated enough to allow the embedded engineer to concentrate on the end product's functionality. The days of building the tools followed by the application are over.

Another offshoot of the Internet revolution is the proliferation of knowledge. Every RFC (request for comment) is available for immediate viewing via the Internet. In addition, applications relevant to particular RFCs are also available. Every day, companies like iReady are taking the rudimentary elements of the Internet defined by the RFCs and packaging them in silicon. The S-7600A Internet data pump is one good example of this. It's a lot of TCP/IP and PPP functionality confined to a small space. Last time, I showed you how to employ the iReady and Microchip parts on the S-7600A/PIC16F877 Internet Engine as a cigarette pack-sized HTTP web server. This time, using the same PIC-based hardware, I'll show you how to electronically say "The check's in the mail."

E-MAIL 101

Casual Windows users have few choices. E-mail is done with a graphical mail program like Microsoft Outlook or something similar. That is, the mechanics of e-mail are hidden

Command	Syntax
HELO	<SP> <domain> <CRLF>
MAIL	<SP> FROM:<reverse-path> <CRLF>
RCPT	<SP> TO:<forward-path> <CRLF>
DATA	<CRLF>
RSET	<CRLF>
SEND	<SP> FROM:<reverse-path> <CRLF>
SOML	<SP> FROM:<reverse-path> <CRLF>
SAML	<SP> FROM:<reverse-path> <CRLF>
VERFY	<SP> <string> <CRLF>
EXPN	<SP> <string> <CRLF>
HELP	(<SP> <string>) <CRLF>
NOOP	<CRLF>
QUIT	<CRLF>
TURN	<CRLF>

Table 1—One good thing to say about the stuff on the Internet, the folks who implement constructs like SMTP try hard to make it all make sense. For instance, TURN says "Let's turn it around—I'll receive and you send." Just like kids on the playground.

behind a pretty and predefined package. This is a good thing, seeing that the casual e-mailer doesn't need or want to know what's going on behind the scenes. He or she simply wants a message to be delivered to the recipient at a specified address. A Sendmail DLL exists that allows Windows developers or serious Windows users to mail by wire. And, as you'll see a little later on, there are commercially available and careware terminal emu-

lators for guys and gals who want to know.

Unix and Linux users have it a little better if you consider more options and more complexity a better thing. Like the Windows user, the Unix and Linux user can use a canned mail program to send and receive messages via a mail server. Users of these systems also have the capability of writing their own programs using languages like Perl or shell scripts to send and receive mail the old-fashioned way. And, somewhat like Windows, the Unix community has a mail program interface that allows mail servers and mail clients to be created.

In either case, if there's a will, there's a way. Users of Windows- and Unix-based operating systems can use and abuse applications like Telnet or FTP to gouge their way through an e-mail transfer. There's no reason to go in that direction because e-mail is really a simple and structured process that is, like most everything that works on the Internet, based on standards. In the case of e-mail, RFC 821 is the standard document for people interested in building simple e-mail

devices. The official name behind e-mail and RFC 821 is Simple Mail Transfer Protocol, or SMTP. RFC 822 and no less than five updates are also part of the e-mail specification. I'll focus on the concepts in RFC 821 and some of the language in RFC 822. Basically, I'm warning you that I won't be attempting to cross-reference the various RFC updates in this article, and this is by no means intended to be a doctoral thesis on e-mail. I simply want to show you how to manipulate the e-mail mechanism, enabling you to send e-mails with small embedded devices.

The whole idea behind SMTP is to make mail delivery reliable and efficient. SMTP really doesn't care what buggy it rides on, but the buggy had better have good wheels and a horse with a sense of direction. In other words, SMTP depends on a reliable datastream like TCP/IP. Because SMTP is independent of the transport, it can work over networks of various machine types in the same room or networks of different machine types between continents.

SMTP is based on a sender/receiver relationship. The sender requests and establishes a two-way communications channel to the receiver. The SMTP receiver can be the ultimate destination or just one hop along the way. The sender is in control as commands are generated by the sender and acknowledged by the receiver. After the communications channel is open, a lock-step process is executed between the SMTP sender and receiver.

After successful connection and host verification, the first command from the sender is a MAIL command. If the receiver can handle the sender's mail, it responds with an OK reply code. The MAIL argument specifies who the mail is from. This argument is called a reverse-path. That is, any messages, good or bad, can be sent via the reverse-path to the sender. Normally only bad messages make their way back to the sender this way.

The MAIL command is followed by a RCPT command. RCPT is short for recipient or the final intended receiver of the mail message. RCPT is

Reply	Function
211	System status, or system help reply
214	Help message (Information on how to use the receiver or the meaning of a particular non-standard command. This reply is useful to you.)
220	<domain> Service ready
221	<domain> Service closing transmission channel
250	Requested mail action OK, completed
251	User not local; will forward to <forward-path>
354	Start mail input; end with <CRLF>.<CRLF>
421	<domain> Service not available, closing transmission channel (This may be a reply to any command if the service knows it must shut down.)
450	Requested mail action not taken: mailbox unavailable (e.g., mailbox busy)
451	Requested action aborted: local error in processing
452	Requested action not taken: insufficient system storage
500	Syntax error, command unrecognized (This may include errors such as command line that's too long.)
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command parameter not implemented
550	Requested action not taken: mailbox unavailable (e.g., mailbox not found, no access)
551	User not local; pleas try <forward-path>
552	Requested mail action aborted: exceeded storage allocation
553	Requested action not taken; mailbox name not allowed (e.g., mailbox syntax incorrect)
554	Transaction failed

Table 2—These numbers weren't just pulled out of the air. Every digit has a particular meaning. Again, like most legacy Internet content, a lot of thought results in a simple, logical, and understandable solution.

Reply	Function
500	Syntax error, command unrecognized (This may include errors such as "command line too long.")
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command parameter not implemented
211	System status, or system help reply
214	Help message (Information on how to use the receiver or the meaning of a particular non standard command. This reply is useful only to you.)
220	<domain> Service ready
221	<domain> Service closing transmission channel
421	<domain> Service not available, closing transmission channel (This may be a reply to any command if the service knows it must shut down.)
250	Requested mail action okay, completed
251	User not local; will forward to <forward-path>
450	Requested mail action not taken: mailbox unavailable (e.g., mailbox busy)
550	Requested action not taken: mailbox unavailable (e.g., mailbox not found, no access)
451	Requested action aborted: error in processing
551	User not local; please try <forward-path>
452	Requested action not taken: insufficient system storage
552	Requested mail action aborted: exceeded storage allocation
553	Requested action not taken: mailbox name not allowed (e.g., mailbox syntax incorrect)
354	Start mail input; end with <CRLF>.<CRLF>
554	Transaction failed

Table 3—If you didn't see the logic in it before, breaking down the digits should bring out the Spock in you.

termed a forward-path. Again, the receiver checks to see if it can accept the mail for the recipient, and if it can, another OK code is returned.

More than one recipient can be specified, and when all of the recipients are good to go, the mail message is transmitted to the SMTP receiver behind the DATA command. The mail message is delimited in a special way. A single period on a line by itself signifies the end of the message. If all goes well, the mail is sent along its way and the SMTP receiver issues a final OK return code. At this point, the connection between the SMTP sender and receiver is normally closed.

The return and OK codes I mentioned are actually numeric codes sometimes followed by an explanatory text message. The numeric part of the reply is intended for the machines, and the text content of the reply is for human consumption. The text makes it easy to debug an e-mail session without having to pull out the scope or read a dump. The SMTP commands and return codes, or replies, are defined in the SMTP standards. Commands and replies are not case-sensitive. A list of the commands and their syntax per RFC 821

is shown in Table 1.

And, Table 2 shows what the replies look like as of RFC 821.

It's a bit overwhelming at first glance, but the truth is that you only need five of the commands to successfully send e-mail. And, if you're doing things right, you'll only encounter three of the replies. One of those commands you haven't been formally introduced to is HELO. HELO is like Mick's line in the song "Sympathy for the Devil." You know, "Please allow me to introduce myself..."

When the sender and receiver open a communications channel, an exchange between the hosts occurs to verify who's who. As you have ascertained, HELO is short for hello. HELO and the domain argument are sent by the SMTP sender to identify the sender's domain. The receiver replies with another line from Mick's song, "Pleased to meet you; hope you guessed my name." (Hey, the name Muhammed Ali is used in an RFC

822 example, so I can use Mick in mine can't I?)

Now that you kind of know what to expect in terms of command and reply, this is a good spot to show you how an SMTP session works graphically. Photo 1 is a screen shot of an SMTP session that I invoked manually using a careware terminal emulator called EasyTerm. You can read about careware and get a copy of EasyTerm by visiting www.arachnoid.com. I'll follow the example session in RFC 821, and at the end, should I be successful, an e-mail message will be generated and sent.

Referencing the commands and replies I listed earlier, reply 220 is usually a good thing. The first line of the manual e-mail session is a 220 OK reply returned for a good TCP/IP connection between my EasyTerm terminal and the mail port (0x19 or 25 decimal) at my ISP cfl.rr.com. I then entered HELO cfl.rr.com and hit Enter. The 250 reply is the high sign for the HELO command. As you can see in the text of the 250 reply, the receiver identifies itself to the sender here. This sequence of HELO commands and replies is an indicator that no transactions are in progress, and the e-mail process is in the cleared and idle state as far as buffers and state tables are concerned.

The MAIL FROM: command and its argument, which was also entered manually, starts the process that puts mail data in a mailbox. The MAIL command also clears the forward- and reverse-path buffers and inserts the

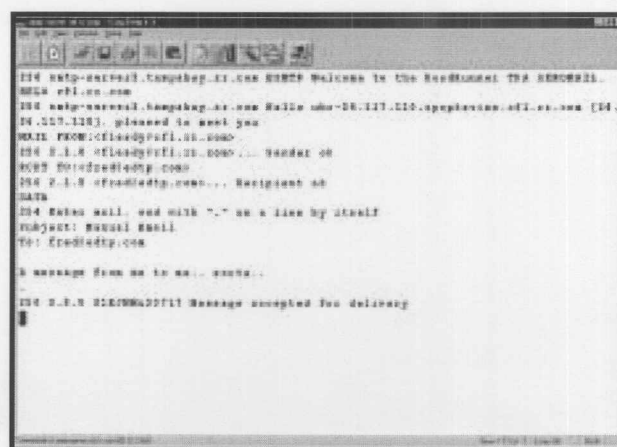


Photo 1—This is pretty simple. But then again, that's the whole idea.

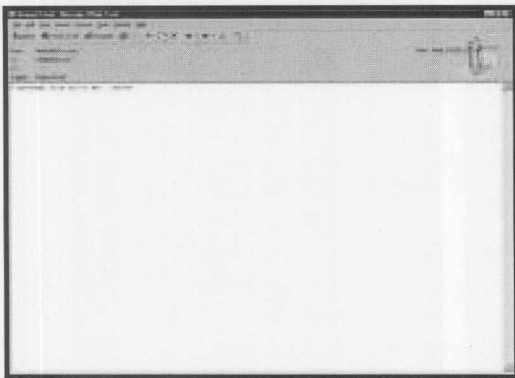


Photo 2—Having a couple of ISPs is great if you write articles for a living like I do. The fact is, you can send e-mail to yourself from yourself all day using your ISP's mail server.

argument (in this case, fleady@cfl.rr.com) in the reverse-path buffer. That means any error messages will be returned to fleady@cfl.rr.com. Another 250 reply tells you that the sender is valid.

If you're wondering why there are two OK replies, 220 and 250, the story lies in the numbers themselves. The most significant digit, 2, means that the message is a positive completion reply. If the middle digit is also 2, then the reply has to do with a connection or the communications channel. A middle digit of 5 references the mail system. The third digit gets into specifics of what the first two digits are coarsely defining. A good set of replies to look at includes the 220, 221, and 421 replies. To give you an idea as to how the replies use numbers to group themselves, I've included the reply list (as of RFC 821) sorted by functional group (see Table 3).

Getting back to the manual e-mail session, the next command I entered is RCPT and its argument, the destination address of the mail data. There can be bunches of RCPT command lines, and again, 250 is the reply you want. The argument of the RCPT command is put into the forward-path buffer.

The next command I entered is DATA. After the 354 is returned, everything in the ASCII character set that follows this command is buffered in the mail data buffer. An end-of-mail indicator, <CRLF>.<CRLF>, is appended to the text after the message body. After the end-of-mail indicator is

sensed, all the data in the forward-path, reverse-path, and mail data buffers is used to get the mail from point A to point B. If everything goes as planned, the mail is forwarded, all of the buffers are cleared, and a 250 OK reply is returned.

There are a couple of things missing in Photo 1. The first missing item is the fifth command, QUIT. Every command must have a single reply. So, the last missing element is the 221 reply from the receiver.

After that, it's lights out. All of that manual labor paid off as the e-mail you see in Photo 2.

Note the "Subject:" and "To:" in the message after the 354 reply. This is an optional header you can insert. It is followed by a blank line, which separates the header from the message body. This format is some of what you'll find in RFC 822. Leaving out the header would not have put any text in the received e-mail note's "Subject:" line, and the RCPT argument would have replaced the received e-mail note's "To:" line text.

I have by no means delved deeply into the bowels of SMTP. I suggest looking deeper at RFC 821 and RFC 822 if you require further knowledge about the commands and replies. These RFCs are good reading, and if you really want to drive the SMTP process hard, there's plenty of content there to help you succeed.

AUTOMATIC TRANSMISSION

You don't know it yet, but you've graduated. You have a solid knowledge of how to send a simple e-mail

by hand. To make the e-mail trick work on the S-7600A/PIC16F877 Internet Engine, all you have to do is a knowledge transfer.

From my previous articles, you already know about the S-7600A/PIC16F877 Internet Engine hardware and bootloader and how they work together. So, you can do a big GOTO in the source code you see in Listing 1(ReferenceSoftware).

Let's start at the comment // SMTP code begins here. All of the code up to this point has been gathering information needed to connect, and then using that information to establish a PPP connection to my ISP. To connect to your ISP, simply put your ISP's IP address in place of mine in the source code. The destination port is already set for 0x19, the well-known SMTP port. Your ISP phone number, log on ID, and password are prompted for in the beginning of the program.

Just like Photo 1, the first thing that should happen after a successful connection is the first 220 howdy message. Because I'm originating from a dial-up connection to ddi.digital.net, my HELO command contains an argument of ddi.digital.net. I only need to check the first four characters of any reply to know if things are good or bad. Actually, the first three would suffice, but it was a bit more logical for me to include the space following the reply to be absolutely sure I had parsed the reply code. One RFC says you can do it with a single character, and another tells you the space after the three digits is important. So, I took both RFCs' advice. I still check the first character, and I also look for the space.

Like the IP address, you'll need to change the MAIL and RCPT arguments to match your mail's source and destination. By now, you've got the idea. The PIC program is no more than a copy of the manual process I performed earlier. Only this time, the PIC is doing the keyboard work. All you have to do to get going for yourself is to literally fill in the blanks in the code I've provided for you for download.

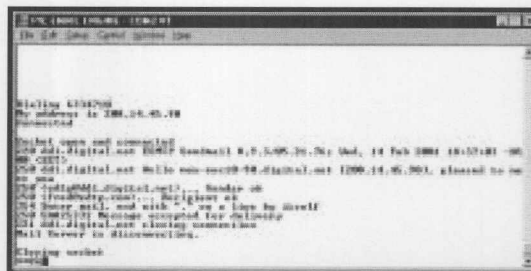


Photo 3—This is the script that will play out when you run the code in Listing 1. Every line that begins with a three-digit reply code is generated by the receiver or mail server.

MAIL DELIVERED

As you can see from the code, the real fun begins after the DATA command. The idea here is to spark your imagination. In the message text, I've pretended that the flood detector in the basement has detected an inch or so of water on the basement floor. I've also made your day by telling you that someone is probably in your garage and it isn't you because you're somewhere else reading this e-mail. What I'm saying is that any event you can capture with a microcontroller can be used to trigger an e-mail message. What's really nice is that you don't have to write any complicated code to convey your message. Photo 3 is the actual Tera Term Pro window that captured the e-mail process that produced the e-mail message in Photo 4.

After you've run my example and understand the mechanism, you can expand on the concept easily. In the previous two articles, I supplied you with a wealth of PIC16F877 and S-7600A utility routines that enable you to:

- get data via a menu or file transfer for entry into internal and external EEPROM
- initialize the Dallas DS1629 real-time clock via menu
- read time and temperature data from the DS1629
- read and write internal and external EEPROM
- use the TCP/IP socket functionality of the S-7600A
- use the PPP functionality of the S-7600A
- control a modem
- interface to Tera Term Pro

You now have the capability of taking the individual routines and applying them to your final e-mail program. For instance, you can store individual e-mail messages in EEPROM and retrieve the one you want to send depending on what triggers the e-mail process. You could have an "it's too hot message" that is triggered by the DS1629 or a "door is ajar" message picked up by the PIC's I/O pins attached to simple micro switches.

Basically, if you can sense it, you can send e-mail to anyone about it.

Now that you are an embedded e-mail guru and we can all keep in touch, I'll finish up the installation of the EDTP embedded device server, which will reside at 216.53.172.209. I'll use this server next time to demonstrate interactively how to add yet more networking capability to tiny embedded devices like the S-7600A/PIC16F877 Internet Engine. By interactively, I mean that you'll be able to use the power of the Internet to test-drive the technology you see in my articles.

You know how to serve web pages using a PIC. You know how to send e-mail using a PIC. When you're finished reading the next installment, you'll know how to configure and implement a PIC-based TCP/IP client. I'll even throw in some words on the server end too. See you then! ☺

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

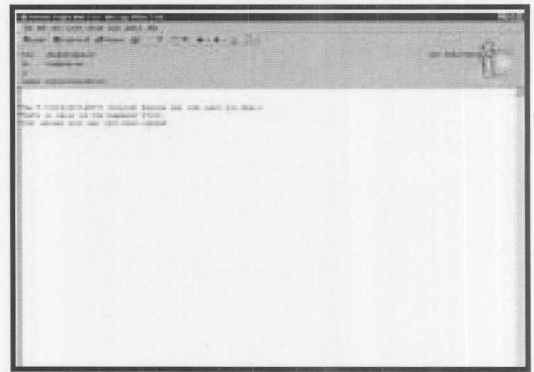


Photo 4—A well-placed switch in an office chair could send the sitter a message every time he or she sat down. If you do this, have a good place to hide.

SOFTWARE

<http://www.chipcenter.com/circuitcellar/march01/ancil-0301/co301fel1.htm>

RESOURCES

J.B. Postel, RFC 821,
<http://www.sendmail.org/rfc/0821.html>.
D.H. Crocker, RFC 822,
<http://www.sendmail.org/rfc/0822.html>.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.

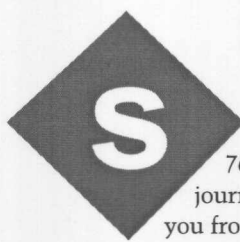
FEATURE ARTICLE

Fred Eady

An S-7800A/PIC16F877 Journey

Part 3: Road Testing

Using his signature music analogies, Fred explores practical extraction or report language, also known as Perl. Just as Janis Joplin grew in popularity, so has been the journey of Perl, with its talent for encoding messages. We've gone from web services to the post office and it's always good to end a series with a little music. Cue the final song.



So far, this S-7600A/PIC16F877 journey has taken you from the land of web services to the post office and all of this has been accomplished without modification to the mainline hardware design. This is the final leg of your trek. In this segment, I'll show you how to connect your S-7600A/PIC16F877 Internet Engine to other hosts. In the process, I'll also demonstrate how to pass data from one host to another with one of those hosts being the Internet Engine.

JANIS JOPLIN SCHOOL OF PROGRAMMING

Well, the late great Janis used to be known as Pearl (not Perl, but close enough). Having little widgets like the S-7600A/PIC16F877 Internet Engine is nice, but they sometimes need to convey information to others who are a bit larger in stature and more powerful for processing. For instance, assume an Internet Engine is monitoring the number of people who enter a small store in a shopping mall. Each time someone enters, the Internet Engine adds the extra body to

the total count. At closing time, the Internet Engine picks up the phone and calls in the total. To be sure that the count was accepted, the Internet Engine waits for an acknowledgment. You already know how to implement on the Internet Engine side, but what about the remote host side? Assuming the remote host is not another S-7600A/PIC16F877 Internet Engine, that's where Perl comes in.

There's no doubt that many of you in the audience could whip out a C, VB, or assembler program to fill the server side of this equation. That's nice, but Perl has built-in socket capability to make the job quick and easy. I won't be counting bodies in this example, but I will be transferring data to a Perl-equipped Linux host attached to the Internet at 216.53.172.209. You can use the example code I will provide to talk to the very same Linux host via your Internet Engine and the Internet from the comfort of your lab or living room.

WHO OR WHAT IS PERL?

Sure, we all know Janis and remember the intense songs she belted out with Big Brother and the Holding Company. That's a pearl of a different kind. The Perl you'll be dealing with is an acronym. Perl is four letters that stand for practical extraction and report language. Larry Wall spawned the Perl language while trying to produce some reports from a Usenet news-like hierarchy of files for a bug-reporting system. Larry's objective was to create a tool he could use again. When he was done crunching code, Perl was born.

Like most other good stuff used on or by the Internet, Perl was introduced to the masses by way of Usenet. And, of course, Perl grew in popularity and was enhanced with features requested by folks on the Internet. This resulted in the Perl you have access to now. My distribution

of Linux includes Perl version 5.6.0.

Perl is a go-between shell and heavy-duty C programming. Using Perl eliminates many of the hassles associated with coding in C and assembling shell scripts. Rather than write a GUI-based C or VB program for the server side of your S-7600A/PIC16F877 Internet Engine project, I decided to use Perl. Why? Well, because it was readily available on my Linux box and I could mash together some Perl code and test it quicker than doing it in C. Another reason I decided to use Perl is because it is self-documenting. At a glance, any of you can pick up the gist of a well-written Perl program. If you think everything that deals with Internet communications needs to be done in C, check out Liu Kai's application of Perl in his article, "Using Perl in Embedded Software Development," Circuit Cellar Online, March 2001.

SETTING UP THE SERVER

I've casually mentioned clients, hosts, and servers so far without going into depth as to who is who. The good news is that it really doesn't matter as far as the user goes. For the programmer, it is self-defining. The server is the host that runs the Perl program that listens for a TCP/IP connection request. The client code initiates that request.

This is a simplistic way of looking at the client/server model. Using Perl, the coding is almost as obvious. To make good use of Perl's capabilities in this area, the programmer must understand the basic building blocks of the Internet. One of those building blocks is the socket.

A socket is a communications endpoint that can provide reliable or unreliable communications services. To a programmer, a socket is a standard structure containing essential elements needed to communicate with other programs that may or may not be running on the same host.

Reliable and unreliable sockets are tied to other Internet building blocks called streams and datagrams, respectively. Properties of stream-based sockets make them reliable, bidirec-

tional sequenced communication building blocks. On the antimatter side of that, datagram-based sockets lack properties that would guarantee sequenced or reliable data delivery. When used on top of Internet Protocol (IP), streams are associated with TCT (Transmission Control Protocol), and datagrams use UDP (User Datagram Protocol).

In addition to an IP address, the socket type (stream or datagram) is also dependent on which of the two aforementioned protocols will be used. One other component, the domain, helps to define the nature of a socket. A domain can be defined as a Unix or Internet domain. In Perl, the Internet domain is denoted as PF_INET and the Unix domain is referenced as PF_UNIX. I'll only travel in the Internet domain and my socket types will be designated as SOCK_STREAM or SOCK_DATAGRAM. To keep complexity to a minimum, I can tell you now that I won't be using UDP in my example, so the SOCK_DATAGRAM type won't be used.

In Perl, the built-in socket and IO::Socket modules support these socket definitions. The Perl IO::Socket module is an object-oriented approach to socket programming and is built on properties of the standard socket module. This approach makes life easier for the socket programmer because he or she doesn't have to tiddle every little bit along the way. The socket programmer can now call the original socket module functions using the less complicated IO::Socket module calls. Keeping with the dual domain standard, both Unix and Internet domains are represented as classes of the IO::Socket module. The class I will draw from is IO::Socket::INET. As you've already guessed, the UNIX domain programmers would use IO::Socket::UNIX. For those of you who enjoy control, the standard socket module functions are also supported by Perl and can be used instead of the high-level IO::Socket function calls. Generally, you have a bit more control over things if you use the standard socket programming techniques. If you don't

have to or don't want to tweak, the IO::Socket methods will do just fine. Guess which one I'm going to use. Let's compare the same server socket formation process performed with the standard socket calls versus the IO::Socket calls.

There are three major steps in creating a server socket:

- call the socket function to make the socket
- bind a name (port number and IP address) to the socket
- call/listen to wait for a connection request

Here's the standard Perl coding for creating a stream socket in the Internet domain:

```
use Socket;
socket(FH, PF_INET, SOCK_STREAM,
      getprotobyname('tcp')) || die
$!;
```

The first line indicates that you are using the library of standard socket services provided with Perl. The second line is the socket call complete with arguments. The first argument, FH, is the name of the file handle the socket will be associated with. The file handle receives requests from clients. Each specific connection is passed to a different file handle by the accept function. I'll cover this process when we get there. You recognize the second argument as the network domain identifier for the Internet domain. The domain definition also tells you that IP addresses, not files, will be used for connections. Recall that SOCK_STREAM tells you that the socket is of the type stream, which implies TCP/IP will be the protocol used by the socket. The last argument is actually a number that represents the protocol type. The function gets the protocol number represented by "tcp" and passes the number to the socket function.

The reason for using a function to return the protocol number instead of just plugging in the right number for TCP/IP is simple. The numbers change. So, to be sure the right protocol number is passed, a call is made to

retrieve the current protocol number that is embedded in the library. Creating a socket is a do-or-die situation. If an error occurs, the socket creation function dies and sends an error message via \$!. \$! is the standard error handling device for Perl. A message and an error number are associated with \$!. If all goes well, that's all I should have to say about that.

The next step is to bind the server socket to a port on the local machine by passing a port and an address data structure to the bind function via sockaddr_in. The Perl code for this operation is:

```
use Socket;
$socket_name = sockaddr_in(80,
    INADDR_ANY);
bind (FH, $socket_name) || die
    $!;
```

Line 1 remains the same here as in the socket creation example. The second line above allows the system to pick an appropriate IP address (INADDR_ANY) and designates well-known Port 80 (HTTP) as the port of choice. FH references the socket I just created in the preceding code. Again, the bind function is do-or-die resulting in error messages being generated using the built-in services of \$!.

The final step in the server socket creation process is to invoke the listen function. Kicking off the listen function signals the operating system that the server is ready to accept incoming network connections on the designated port. The listen function is simple:

```
listen (FH, $length)
```

In the listen function arguments, the socket is again represented by its file handle. To accommodate multiple clients, a queue length is specified. The queue length is the number of clients that can wait for an accept at one time.

I promised to discuss the accept function when the time was right. Well, it's right. When a client requests a connection, the accept function makes the connection and assigns a new file handle specific to that con-

nection. The coding for the accept function is shown below:

```
accept (NEW, FH) || die $!;
```

The new file handle that accept assigns is represented by NEW in the argument field. The answering socket file handle, the original server socket, is the last argument before the do-or-die procedure. At this point, the server can read and write to the file handle new for its communication with the client.

OK. Now let's do the same thing with IO::Socket:

```
use IO::Socket;
$sock = new IO::Socket::INET
    (LocalAddr => 'edtp.com',

    LocalPort => 8080,

    Proto    => 'tcp',

    Listen   => 5);
die "$!" unless $sock;
$new_sock = $sock->accept();
```

You already know what the first line does. The rest of the code short of the do-or-die creates a socket, binds the socket with an IP address and port number, and invokes the listen function with a queue length of 5. Everything is in place and just waiting for a connection request from the

client. When a connection request is received, the accept method is called and a new socket is created to service the request. In both the socket and IO::Socket examples, a close must be issued to destroy the socket structure when communications is complete.

APPLYING WHAT WE'VE LEARNED

While it's fresh, let's put together the Perl code to accept some data from the Internet Engine and spit it right back out. I'll choose an arbitrary port number of 8080 decimal and use TCP/IP as the protocol and delivery method. When the socket is created, I'll allow up to 20 Internet Engines to wait in the queue for a connection. When an Internet Engine connects successfully, it will send an ASCII string to the server. The server will, in turn, send back or echo the received ASCII string to the Internet Engine. The communications session will end and wait for another Internet Engine to call in and connect. The application is quite simple and so is the code (see Listing 1).

The LocalAddr and LocalPort constructor options provide the IP address and port number that will be used to bind the socket. I promised 20 Internet Engines in the queue and the listen option performs that action. SOCK_STREAM tells you TCP/IP is

Listing 1

```
$linuxsvr = IO::Socket::INET->new (LocalAddr =>
    '192.168.1.200',
    LocalPort => 8080,

    Type => SOCK_STREAM,

    Reuse => 1,

    Listen => 20)
    or die "Server initialization failed for Port 8080\n";

while(1)
{
    $intengine = $linuxsvr->accept();
    $intengine->recv($buffer,256);
    $intengine->send($buffer);
    close($intengine);
}
close ($linuxsvr);
```


Photo 1—You can put up to 256 characters into your Internet Engine code for transmission to the Florida-room Linux server. I sent a four-letter word, my name.

in effect. You haven't been exposed to the reuse option. If reuse is a nonzero number, this constructor option allows the local bind address to be reused should the socket need to be reopened after an error. I've been doing or dying all through this article so there's nothing new in that line of thought.

Moving to the meat of the Perl code, it looks like I really wrote this in C. Not! After the server socket is created, a never-ending loop is entered. The IO::Socket mechanism has already put the server at Port 8080 in Listen mode. The next task is to accept any incoming connect requests, create a new socket object, and process the communications session.

The new socket object receives data from the Internet Engine. This data can be just about anything as long as it is meaningful to the programmer of the Internet Engine. A receive data buffer of 256 bytes is allocated for this process. This simple example does nothing with the data. It simply resends the contents of the receive buffer back to the remote Internet Engine. When the data is sent, the newly created socket object is closed and the server reverts back to waiting for a new connection request. If anything blows up the program, the final close statement kills the server socket.

INTERNET ENGINE CLIENT CODE

I told you in the beginning that the S-7600A/PIC16F877 Internet Engine can serve web pages, send e-mail, and talk to other hosts using standard Internet protocols. At that time, you

probably didn't expect to be able to do all of these things without rewriting the code every time you needed a different function. If you followed the first installments, you know that the code changes have been minimal and most of the code is reused from application to application. Well, there's been no change in that paradigm.

In the main code segment, the changes include bit twiddling to put the S-7600A in Client mode and writing a different port address to the S-7600A registers. The new client module begins at the comment Client Algorithm. Algorithm is a big word for a small amount of code. Basically, the client code is simpler than the Perl server code.

When the Internet Engine fires up the modem and makes a connection to the local ISP, the TCP/IP stack inside the S-7600A is called on to request a connection to the Linux server at 216.53.172.209 on Port 8080 decimal. When the TCP/IP handshaking is done and the connection is established, the Internet Engine sends a canned set of characters to the Linux server's Perl program. It then waits for the letters it sent to return. After it gets them, it prints them to the terminal screen and closes the socket. I sent the letters to spell out my name followed by a carriage return and linefeed. As you can see in Photo 1, it all worked as planned. The complete Internet Engine client code is provided for downloading at <http://www.chipcenter.com/circuitcellar/april01/ancil-0401/c0401fecode.zip>

GET ON THE 'NET

I've put up the Linux server at 216.53.172.290 for Circuit Cellar's readers. There will be lots of goodies at this address. The application you've just read about is at Port 8080. I'll post an address book on www.edtp.com to let you know what's on the Linux server and how to access it. Also, I've received lots of requests for assembled Internet

Engines. I am now offering the S-7600A/PIC16F877 Internet Engine assembled, tested, and loaded with the boot loader code.

Writing this series of Internet Engine articles has been great fun. I hope to see you echoing messages from your site very soon and I hope I've proven beyond a shadow of a doubt that it doesn't have to be complicated to run on the Internet. ☺

Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.